IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 31, 2020, accepted June 11, 2020, date of publication June 15, 2020, date of current version June 24, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3002591

Uvis: A Formula-Based End-User Tool for Data Visualization

MOHAMMAD AMIN KUHAIL[1] **AND SOREN LAUESEN**[2]
[1]College of Technological Innovation, Zayed University, Abu Dhabi 144534, United Arab Emirates
[2]Computer Science Department, IT University of Copenhagen, 2300 Copenhagen, Denmark

Corresponding author: Mohammad Amin Kuhail (mohammad.kuhail@zu.ac.ae)

This work was supported in part by the Danish Research Council's Nabiit Programme, and in part by the Zayed University, United Arab Emirates.

ABSTRACT Existing approaches to data visualization are one of these two: accessible to end-user developers but limited in customizability, or inaccessible and expressive. For instance, commercial charting tools are easy to use, but support only predefined visualizations, while programmatic visualization tools support custom visualizations, but require advanced programming skills. We show that it is possible to combine the learnability of charting tools and the expressiveness of visualization tools. Uvis is an interactive visualization and user interface design tool that targets end-user developers with skills comparable to spreadsheet formulas. With Uvis, designers drag and drop visual objects, set visual properties to formulas, and see the result immediately. The formulas are declarative and similar to spreadsheet formulas. The formulas compute the property values and can refer to data from database, visual objects, and end-user input. To substantiate our claim, we compared Uvis with popular visualization tools. Further, we conducted usability studies that test the ability of designers to customize visualizations with our approach. Our results show that end-user developers can learn the basics of Uvis relatively fast.

INDEX TERMS End-user development, formulas, visualization tools, information visualization.

I. INTRODUCTION

Data visualization aims at supporting human abilities by showing data using visual variables such as position, color, size, and orientation [1]. The insights of data visualization can be applied in many fields such as healthcare, finance, and agriculture.

Charting tools allow designers to build a visualization by selecting predefined visualization templates and mapping them to data. Designers can adjust the visual properties to a limited extent. This approach improves *learnability* (learning how to build a visualization) as well as *task efficiency* (building it fast). However, it lacks *expressiveness* (customizing the visualization to specific needs). Data analysis tools such as Tableau and its predecessor Polaris [2] integrate well with existing data and help users explore the data. They do not require programming skills. However, like charting tools, there is no way to create visualizations beyond what is predefined. Visualization tools such as Lyra [5], iVisDesigner [6], and Data Illustrator [24] allow the creation of custom visualizations without real programming. These tools are more flexible than charting tools. Nevertheless, they are limited in expressiveness. For instance, they use predefined layouts and visual properties that cannot refer to other visual properties or end-user input. Further, it is not possible to create interactive visualizations with these tools.

Designers may resort to programmatic visualization tools and libraries such as D3 [3] and Vega [12] to accomplish their objective. While such tools offer high visualization expressiveness, they require programming skills accessible to technical audience.

Despite the diversity of existing visualization tools, there is still a gap between tools that are accessible to end users, but limited in expressiveness, and tools that are expressive but only accessible to professional programmers.

We contribute Uvis, a visualization tool aimed at *end-user developers* who have development skills comparable to spreadsheet formulas, but no training in programming. Uvis allows designers to drag and drop visual objects and specify declarative formulas for the visual object properties. A formula computes the value of a property, and can refer to data from databases, visual objects, and end-user input.

To assess expressiveness, we created a collection of visualizations. Furthermore, we compared Uvis with popular

The associate editor coordinating the review of this manuscript and approving it for publication was Gianmaria Silvello.

visualization tools such as D3 [3] and Vega [12]. To assess learnability and efficiency, we conducted usability studies with designers. Our results show that Uvis has high expressiveness, and its basic principles can be rapidly learned by designers with IT skills akin to spreadsheet formulas. Part of this work is based on Kuhail's thesis from the IT University of Copenhagen [27].

In a previous work [33], we proposed designing visual objects for construction of time-oriented visualizations. The work covered a small part of Uvis principles, and only focused on expressing time-oriented visualizations. Further, the work was not mature enough to be evaluated with users and to be compared with other tools such as D3 and Vega.

II. RELATED WORK

We divide approaches to data visualization and user interface design into two categories: tools aimed at non-programmers (charting tools, data analytical tools, and visualization tools), and tools for designers with programming skills (visualization toolkits and programming languages).

A. NON-PROGRAMMER TOOLS

1) CHARTING TOOLS

Charting tools such as Microsoft Excel, RAWGraphs [40], Flourish [42], and Infogram [43] allow designers to create visualizations with predefined templates. Limited customization is possible. For instance, designers can change appearance properties such as color, text formatting, etc. This approach is accessible to non-programmers, but does not support custom visualizations. Designers do not have sufficient control over the building blocks of the visualization. For instance, not all the visual properties of the visual objects are available for modification.

2) DATA ANALYTICAL AND EXPLORATORY TOOLS

Data analytical and exploratory tools such as Tableau [44], Polaris [2], Spotfire [45] and Omni-scope [46] integrate well with existing data and help users explore the data. Despite the expressive power of these tools compared to charting tools, control over graphical output is still limited, making them unsuitable for novel custom visualizations.

3) VISUALIZATION TOOLS FOR NON-PROGRAMMERS

Visualization tools such as Lyra [5], iVisDesigner [6], VisComposer [7], Data Illustrator [24], DataInk [8], and Charticulator [9] allow creation of custom visualizations without real programming. These tools use a visual builder as a development environment [22].

Lyra [5] allows designers to specify visual attributes with mathematical expressions that refer to data fields. However, the expressions are limited. For instance, they cannot refer to other visual properties. Further, it is not possible to create a dynamic visualization with Lyra. For instance, dynamic queries [4] are not supported. To cite an example, it is not possible to add filters that allow for data exploration based on user input.

iVisDesigner [6] allows designers to develop visualizations of complex predefined layouts. Designers can drag and drop visual objects, bind visual properties to data or constant numbers. Despite the power of this approach, the visualization specifications are limited. For instance, designers cannot bind a visual property to a mathematical expression that refers to data and other visual properties. Further, it is not possible to implement interaction beyond what is predefined.

VisComposer [7] allows designers to connect modules together to transform the data. Further, designers can use blocks of code to extend their visualizations with custom behavior. This approach may be accessible to end-user developers, but real programming is needed for high expressiveness.

Data Illustrator [24] uses a ''lazy data binding'' approach. Designers use familiar tools to draw their visualizations without underlying specifications. Later, designers apply data encoding when it is necessary. This approach gives flexibility to designers. However, similar to the other tools, expressiveness is still limited as designers cannot specify visual variables with expressions. Moreover, it is not possible to design interactive visualizations.

DataInk [8] supports creation of visualizations with direct manipulation via direct pen and touch input. Designers design their own glyphs, and map their visual attributes to data. This approach combines the power of data visualization and graphic design tools. Nevertheless, the visualizations produced are static, and expressiveness is limited.

Charticulator allows designers to create charts of different layouts such as parallel coordinates [10] and coxcomb charts [47]. However, like the other tools, it is hard for designers to create a custom visualization with a layout beyond what is predefined. Further, Charticulator does not support interactive visualizations.

In conclusion, these tools are innovative, and designed to be accessible to a wider audience than programmatic tools. However, according to a survey study [31], these tools do not allow high customization. As an example, these tools are not able to make visualizations such as LifeLines [25]. This visualization requires dynamic queries [4], user interaction, showing data from various tables, and showing relationships between data entities.

B. PROGRAMMER TOOLS

1) VISUALIZATION TOOLKITS

Visualization toolkits allow designers to construct traditional and new visualizations by means of domain-specific programming languages tailored for visualization. Examples are Protovis [11], D3 [3], Prefuse [15], and Improvise [16]. The approaches of these tools vary from imperative to declarative programming. However, designers may still need to implement program-like specifications. For instance, designers need to declare variables, program functions, etc. Consequently, the gap between the objective (what the designer

wants to accomplish) and the solution (how the designer accomplishes the objective) remains high. This is described by Norman as the gulf of execution [17].

Vega [12] is a declarative language for creating, saving, and sharing interactive visualization design. Vega does not require developers to write the visualization specifications in a certain sequence. To support interaction, the tool allows developers to write imperative statements as event handlers. To enable custom calculations, Vega uses its own expression language for writing basic formulas. The expressions can refer to data from a dataset as well as event data. However, the expressions have limitations. For instance, they cannot directly refer to aggregate functions (e.g. sum, max). Data transformations must be made first. Further, the expressions cannot refer to visual properties. This limits the designer's ability to create custom layouts by aligning different visual objects in ways beyond what's predefined. Built on Vega, Vega-lite [13] provides a high-level grammar that enables the concise specification of interactive data visualizations. Vega-lite specifications are shorter than Vega. However, it is less expressive. For instance, Vega-lite does not support custom and specific interaction techniques.

Atom [14] is a high-level grammar for *unit visualizations*, visualization where every data item is shown by a distinct visual object (a visual unit). Atom maps each visual object to one data item. Some visual properties such as fill color can visualize data. Atom can express a variety of unit visualizations. However, expressiveness is limited. For instance, visual properties cannot be specified with expressions. Further, position properties are determined based on layout definition. Such limits make Atom unsuited for making data visualizations such as LifeLines [25] where visual objects show relationships between data entities by being aligned to other visual objects showing related data. Besides that, Atom does not support interactivity. Therefore, implementing interaction such as dynamic queries [4] is not possible with Atom.

2) GRAPHICAL LIBRARIES

Graphical libraries such as GDI+ [38] and Java 2D [39] are available for many programming languages. They provide basic components such as polygon, textbox and drop-down box. By means of a program, you can create any visualization, bind to any data and perform any interaction. The program can be integrated with development environments that allow programmers to build a user interface. The environments use the visual builder approach [22]. Programmers manually drag and drop graphical components (buttons, text boxes, etc.) and set their properties. However, programming is needed to make the interface functional.

III. DESIGN

In designing Uvis, we started with a simple idea: Let each visual property be like a spreadsheet cell with a formula. It turned out that with this simple approach, the designer does not need to know programming concepts such as variables, loops, and recursion, allowing designers to focus on piecing together visual objects using formulas. The latest version of Uvis can be found in [37].

This work proposes the Uvis approach that consists of four elements: a development environment, general-purpose visual objects, formulas, and documentation. The *development environment*, shown in Figure 1, consists of six panels: (A) The form being designed, (B) Property grid, (C) Error list, (D) Data map, (E) Toolbox, (F) Visualization specification files, and (G) Data view. All the panels are movable. The visual form (A) contains the visualization the designer is currently building. The property grid (B) allows the designer to change the properties of a visual object. The error list (C) lists the errors in the formulas. The data map (D) shows the structure of the data the designer wants to show. The toolbox (E) is a list of the available visual objects. The visualization specifications (F) are saved as .vis files. The data view (G) shows a sample of the data in the data model. Uvis uses a small collection of *visual objects* including well-known components such as ellipses, bars, splines, pie slices, and labels.

Visual objects can be bound to a data source. As a result, one visual object is created for each row in the data source. However, if a visual object is not connected to data, only one instance of the object is created. A visual property may be constant or a formula.

Formulas are declarative spreadsheet-like expressions. The formulas can refer to data fields, visual properties, and functions.

The Uvis *documentation* is a tutorial that walks the designer stepwise through the main Uvis concepts. It contains various examples. Discussing the documentation is outside of the scope of this paper. The details of the documentation can be found at [48].

In the following subsections, we will discuss the fundamental elements of Uvis: Uvis development environment, visual objects, and declarative formulas.

A. DEVELOPMENT ENVIRONMENT

The Uvis development environment (Figure 1) is a What-You See-Is-What-You-Get (WYSIWYG) visual builder where the designer drags visual objects to the user screen and defines the properties of the components. When the designer changes a property specification, the formulas of all properties are recomputed, and the screen is refreshed. Myers *et al.* [18] gave an overview of user interface tools and explained why visual builders (called interface builders in the paper) were much more successful with local developers than program-based tools. Besides the traditional visual builder features, we incorporated features that support the design process. As an example, when the designer selects a visual object, the property grid shows the properties, formulas, and computed values of the properties (Figure 1B). This allows designers to inspect all properties of the visual object [34]. As another example of assisting designers, we show the data map (Figure 1D) as well as data view (Figure 1G). Since

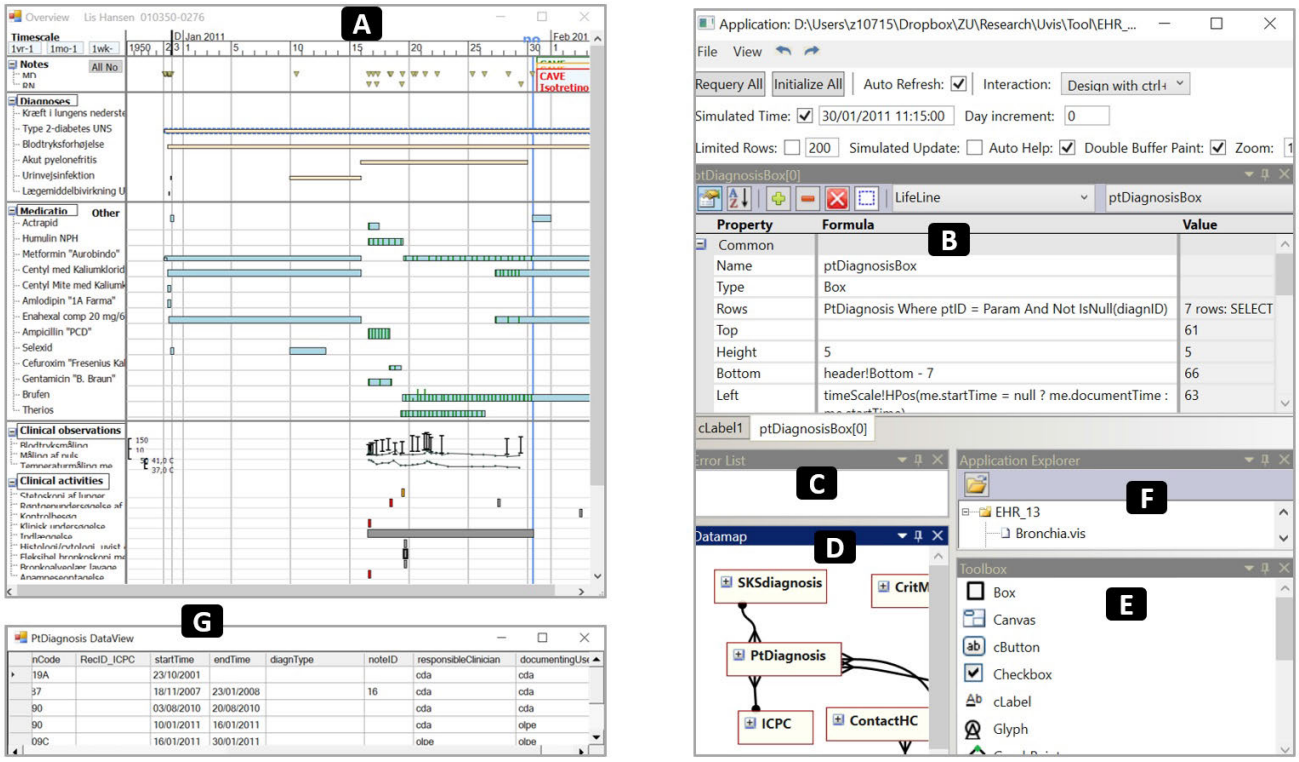


FIGURE 1. Uvis development environment. (A) The form being developed. (B) The property grid of the selected visual object showing the properties and formulas used to specify the object. (C) Error List. (D) The model of the data behind the visualization. (E) A list of visual objects the designer can use to compose a visualization. (F) Visualization Form files. (G) A view of the *ptDiagonsis* table (available upon clicking the ptDiagonsis table).

designers may design a visualization of data coming from several data tables, it is essential to show the data tables and the relationships between them. The model is a traditional Entity-Relationship diagram (E-R diagram) [23]. Admittedly, some designers may not be familiar with E-R diagrams. Uvis documentation [48] explains the principles of E-R diagrams. The Table view panel (G) shows a sample of the data when the designer clicks a table in the data model. Exploring the data in that manner helps designers make sense of data particularly if the data have names that are not self-explanatory. Research showed that novice designers relate to data using concrete values rather than field names [19], [20].

The design of the development environment is the result of usability studies and expert feedback. The details of the usability studies to improve the environment are discussed in [27]. Earlier versions of the environment such as [35] were more primitive.

B. VISUAL OBJECTS

Visual objects are the building blocks of a visualization. Figure 2 shows examples of the visual objects Uvis provides. Unlike many existing visualization tools, Uvis supports standard UI elements (e.g. `Button`, `Textbox`, etc.) that allow for interaction. Furthermore, Uvis includes geometric shapes such as `Triangle`, `Ellipse`, etc. They are inspired by Cleveland [21] recommendations, and can be used to show data as position, color, orientation, etc. Moreover,

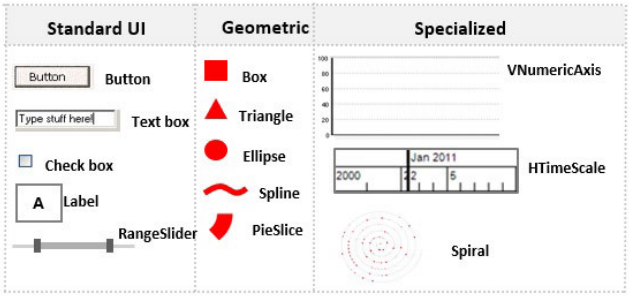


FIGURE 2. Examples of visual objects that Uvis supports.

we designed specialized objects that are commonly used in visualizations. For example, `HTimeScale` is a time scale that shows multiple periods of time horizontally. Designers can let a property refer to a time scale and get the pixel position corresponding to a point in time. Further, the time scale is an interactive object that allows users to navigate through time by dragging the scale. As another example, `Spiral` allows designers to show cyclic time-oriented data on a spiral.

All visual objects have these *common properties*: `Rows`, `Parent`, `Canvas`, `Top`, `Left`, `Bottom`, `Right`, `Height`, `Width`, `BackColor`, and `BorderColor`. The `Rows` property binds visual objects to data. The `Parent` property specifies the parent element. This property allows

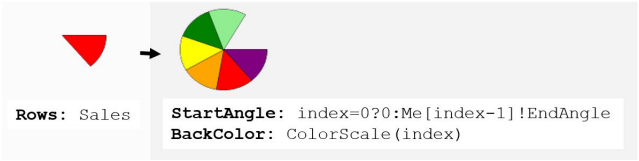


FIGURE 3. Example of a default formula: Setting the PieSlice StartAngle and BackColor.

formulas in child elements to refer to data and visual properties in parent elements. The `Canvas` property specifies the panel on which the object is placed. By default, it is where the designer has dragged the object. `Top`, `Left`, `Bottom`, and `Right` properties position visual objects. `Height` and `Width` are the usual size properties. `BackColor` and `BorderColor` are properties that specify the background and border color respectively. Specialized objects such as `HTimeScale` have *special properties* such as `BorderValues`, which determines the periods of time that are covered by the object. Moreover, designers can add their own *designer properties*. They can compute values that other properties refer to.

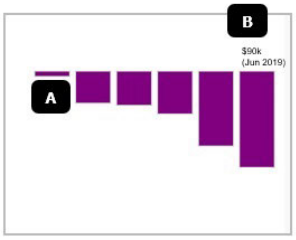
To improve task efficiency without compromising expressiveness, some visual objects provide default formulas that cater for common cases. These formulas are still changeable by the designers if they want a different appearance or behavior. For instance, the `StartAngle` of a `PieSlice` object has a default formula (Figure 3).

C. DECLARATIVE FORMULAS

Uvis formulas are inspired by spreadsheet formulas, which have been successful with end-user developers [18]. Uvis formulas are declarative since they specify what the result of the computation should be rather than how it should be done, and where the result should be stored. Further, they are sequence-free, and do not have loops. Uvis formulas may refer to data fields and visual properties. When a visual object is bound to a data row, the formula refers to the data fields of the row as if they were properties of the component. However, Uvis must be able to handle any field or table name found in the database. A field may for instance be called `BackColor`, which is also a built-in property name. To resolve this ambiguity, Uvis uses dot (.) for data fields and bang (!) for visual properties and other Uvis names. The following subsections will explain Uvis formulas via two examples.

1) UVIS BASIC FORMULAS

Figure 4 illustrates Uvis basic formulas with a bar chart representing sales over the course of six months. This example illustrates a detail-on-demand interaction style [26] as users can see the sales amount as well as the month upon clicking a bar. The bars are made with a `Box` (`SalesBox`). The `Rows` formula binds `SalesBox` to data in the `Sales` table. The data is ordered by the sales amount in ascending order. The result of data binding is that Uvis creates an instance of `SalesBox` for each data row. The `Left` formula calculates



Sales Table

ID	Month	Amount
2	"Feb 2019"	5,000
4	"Apr 2019"	39,000
1	"Jan 2019"	32,000
3	"Mar 2019"	40,000
5	"May 2019"	70,000
6	"Jun 2019"	90,000

Box	SalesBox A
Rows:	Sales Order by Amount
Top:	40
Left:	15 + index*(Width+5)
Width:	45
Height:	Amount/1000
BackColor:	Color.Purple
Click:	AmountLabel!Visible = Not AmountLabel!Visible

Label	AmountLabel B
Parent:	SalesBox
Bottom:	Parent!Top + 2
Left:	Parent!Left
Text:	"$k" & Amount/1000 & NewLine() & "(" & Month &")"
Visible:	init False

FIGURE 4. Creating a basic bar chart with Uvis formulas.

the left position of the bars in such a way that they are spaced 5 pixels apart. The `Height` formula takes the sales amount and divides it by 1000 to get the height. Upon clicking a box, the visibility of a child label on top (`AmountLabel`) is toggled. `AccountLabel` is positioned to be on top of its parent (`SalesBox`) using the `Bottom` property. The labels are initially invisible (`Visible` is set to be false). The `init` keyword indicates that the value is changeable.

2) DATA NAVIGATION AND CONDITIONALS

Figure 5 shows a visualization of patient information inspired by LifeLines [25]. The visualization shows physician notes of patient health and diagnoses mapped to a time scale. The time scale shows three different periods of time of different zoom levels. The data model of the data behind the table is on the top right of the figure. Even for a professional developer, it would be time consuming and challenging to implement such a visualization. With Uvis, it is possible to build it with declarative formulas and two imperative formulas to make it interactive. For space reasons, Figure 5 shows only the interesting formulas.

The `Rows` formula of label `ptLabel` (Component A, Figure 5) connects the label to the record of the patient whose ID is the value of `ptIDText` (Component D, Figure 5). The `Rows` formula of `ptNoteTitle` (Component B, Figure 5) joins the parent table (`Patient` with a selected ID) with the `ptNote` table.

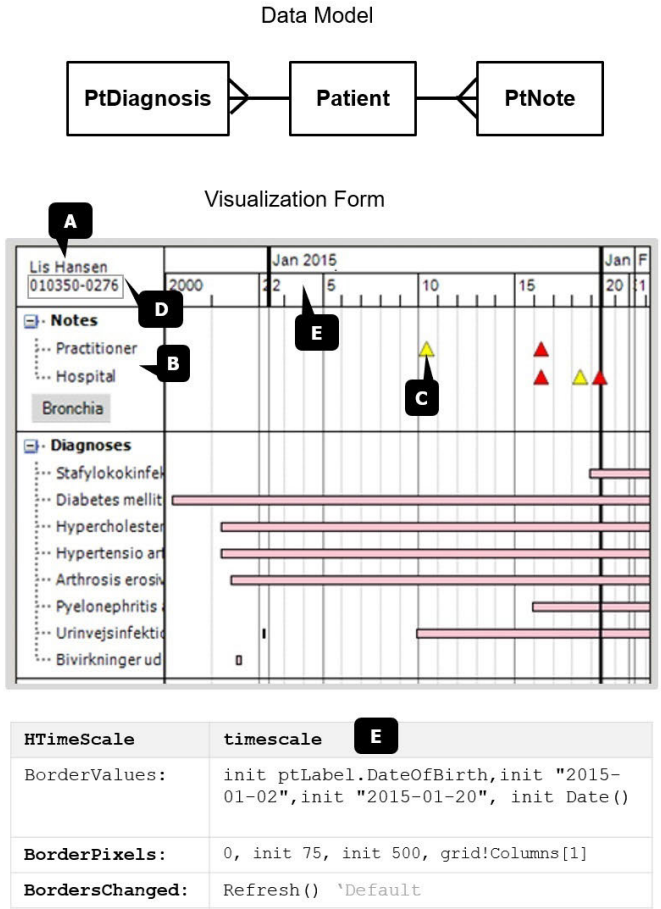
Uvis uses the operators `>-` and `-<` to specify a join between two database tables. `A >- B` is a left join where

Label	ptLabel A
Rows:	Patient where ptID = ptIDText!Text
Text:	Patient.Name

Label	ptNoteTitle B
Parent:	ptLabel
Rows:	Parent-< ptNote Group By Title
Text:	ptNote.Title
Top:	10+index*(Height+5)

Triangle	ptTriangle C
Parent:	ptLabel
Rows:	Parent-< ptNote
Left:	timeScale!Hpos(noteTime)-Width/2
Top:	noteTitle!Top
noteTitle:	Find ptNoteTitle ON me.Title=ptNoteTitle.Title
BackColor:	Choose(Me.noteWarningLevel, Color.LightGreen, Color.Yellow, Color.Red)

TextBox	ptIDText D
Canvas:	grid!Cell(0,0)
Text:	init 010350-0276
FocusLost:	Refresh() 'Default



HTimeScale	timescale E
BorderValues:	init ptLabel.DateOfBirth,init "2015-01-02",init "2015-01-20", init Date()
BorderPixels:	0, init 75, init 500, grid!Columns[1]
BordersChanged:	Refresh() 'Default

FIGURE 5. Building a visualization inspired by LifeLines [25] to show medical information of a patient. Formulas for the following building blocks are shown: (A) A label for displaying patient name (`ptLabel`). (B) Labels for showing patient note titles (`ptNoteTitle`). (C) Triangles for showing notes made by physicians regarding the patient health (`ptTriangle`). (D) A text box for patient ID input (`ptIDText`). (E) An interactive time line for showing the time horizontally (`timeScale`).

we start in `A` and extend it with a matching `B` (nulls if there isn't any). `A -< B` is a right join where we start with B. Uvis also supports inner joins, such as `A =< B`, where rows are included only where there is a match.

We have chosen `-<` and `>-` because they resemble the crow's foot notation in E-R diagrams.

In order to vertically align `ptTriangle` objects (Component C, Figure 5) with the related `ptNoteTitle` objects (Component B, Figure 5), the designer needs to let `ptTriangle` objects find the related `ptNoteTitle` objects. To that end, the designer added a designer property, `noteTitle`, in the `ptTriangle` formula specifications. This property finds the related `ptNoteTitle` objects using the `Find` operator. Aligning the `ptTriangle` objects now is just a matter of setting the `Top` property to be equal to the `Top` of `noteTitle`.

This example shows that Uvis formulas alone allowed for a custom layout due to *cross-referencing*, the ability of formulas to refer to properties of other visual objects.

To make the background color of the `ptTriangle` objects represent note warning, the designer used the `Choose` function known from Visual Basic.

3) USER INTERACTION

To interact with the visualization, the end-user types the patient ID into the textbox `ptIDText` (Component D, Figure 5). When done, Uvis executes the `FocusLost` event handler, which asks Uvis to refresh the screen. As a result, Uvis recomputes the formulas, retrieves the information of the patient with that ID, and displays her information on the screen. This is an example of a dynamic query [4].

As another example of interaction, the user can drag the time scale to the right or the left to focus or get an overview of a specific period of time. This interaction style is inspited by the Visual Information-Seeking Mantra: overview first, zoom and filter, then details on demand [26].

`timeScale` (Component E, Figure 5) shows three periods of time defined by `BorderValues`. The first period covers the duration between the patient birth until the second of January 2015. The second one ends on the twentieth of January 2015, and the third ends today. Each of the periods is shown in a ribbon, specified by `BorderPixels`. By dragging the time to the left or to the right of a time period, the user can change the time the scale is representing. By dragging the time scale borders, the user can change how

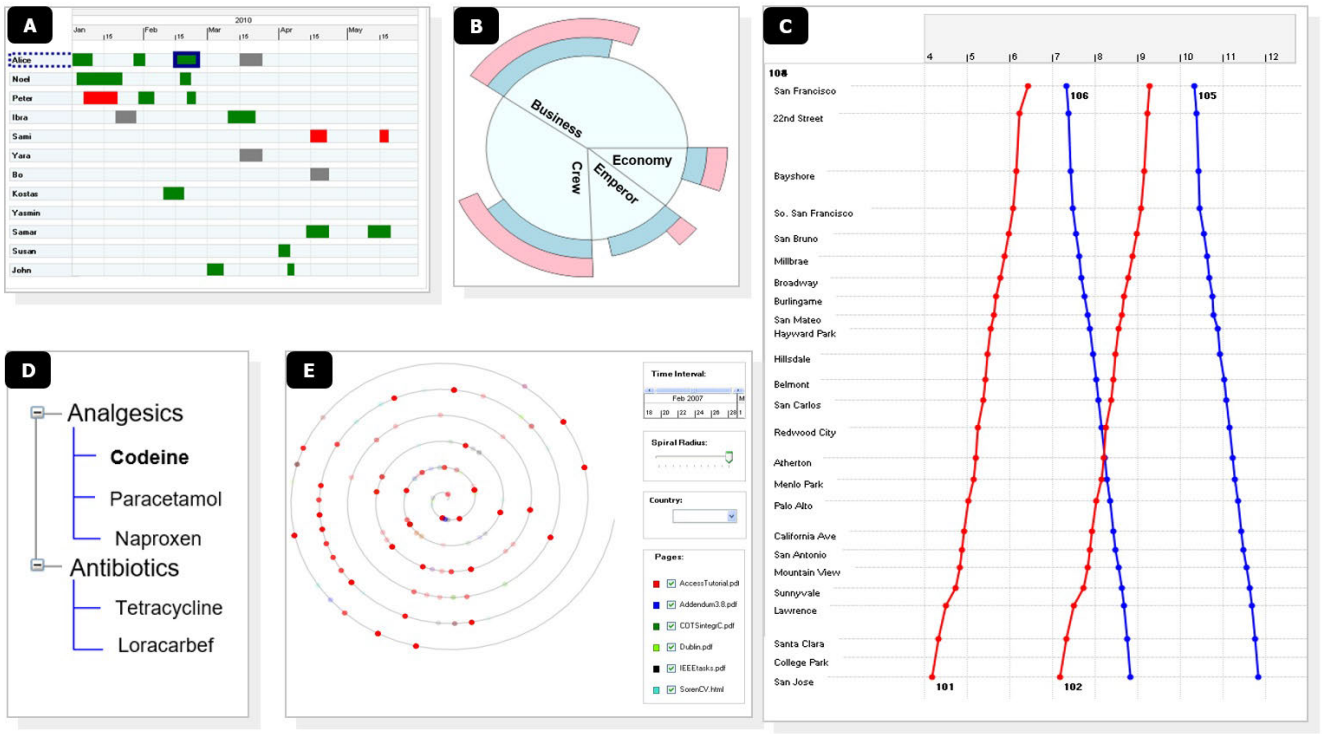


FIGURE 6. Example visualizations built with Uvis: (A) Task plan. (B) Passenger stats. (C) Train schedule. (D) Medicine tree. (E) Spiral graph.

much space each period is taking up. When the user interacts with the time scale, `BordersChanged` is triggered, and as a result, the formulas will be recomputed, and the screen will be updated.

The designer has mapped the time of the patient notes (`ptTriangle` objects) to the time scale using the `HPos` function of the time scale. This function calculates the pixel location of the patient note time. When the user interacts with the time scale, the event handler `BordersChanged` will ask to refresh the screen. As a result, the patient note triangles will be repositioned.

IV. EVALUATION

Uvis was designed to be an expressive, task efficient, and learnable visualization tool. To assess expressiveness, we built a variety of applications. Furthermore, we compared Uvis with popular visualization tools such as D3 [3] and Vega [12]. To assess learnability and efficiency, we conducted usability studies with designers. Our results show that Uvis has high expressiveness, and its basic principles can be rapidly learned by designers with IT skills akin to spreadsheet formulas.

A. EXPRESSIVENESS AND LIMITATIONS

Figure 6 shows example visualizations built with Uvis. The visualizations are fully explained in [27]. Other visualizations can be found in [33]. Table 1 shows an overview of these visualizations. Different data transformations (e.g. filtering, joining, sorting) have been performed to create the visualizations. Further, different types of formulas have been used to create the visualizations. The visualizations show different characteristics and interaction styles. For instance, some have a radial layout whereas others have a linear one. Interaction-wise, some visualizations are based on the details-on-demand metaphor [26], and others allow end-user dynamic queries [4]. The examples are not novel visualizations, but illustrate the expressiveness of Uvis formulas. We have also created other visualizations such as CircleView [29] and Horizon Graph [41]. Some of the visualizations were created using only primitive visual objects. For instance, the visualization inspired by CircleView [29] was created using `PieSlice` objects. Other visualizations such as Horizon Graph required a specialized object (`Area`).

Despite resembling visualizations created with template-based tools such as Tableau, the visualizations in Figure 6 cannot be easily made with template-based tools. Most of the visualizations are based on combining multiple relational tables whereas template-based visualizations are typically based on single datasets. Further, with Uvis, designers have the freedom to customize the interactivity and appearance of the visualizations in a way that is not possible in template-based tools.

Uvis expressiveness depends on four main principles: `Rows` formula, visual property formula, utility functions, and visual objects. Table 2 shows examples of what formulas can refer to.

TABLE 1. Characteristics of selected visualizations made with Uvis.

		Visualizations				
		Task Plan	Passenger Stats	Train Schedule	Medicine Tree	Website Hits
Uvis formulas	**1. Data Transformations** Filter a Table	✓				
	Sort a Table		✓			
	Join Two Tables	✓		✓	✓	✓
	2. Visual Mappings					
	Mathematical positioning	✓		✓	✓	✓
	Data field formulas	✓	✓	✓	✓	✓
	Visual property formulas	✓		✓	✓	✓
	Parent property formulas	✓	✓	✓	✓	
	Conditional formulas	✓		✓	✓	
	Logical formulas	✓				✓
	Sibling formulas		✓		✓	
	Children formulas			✓	✓	
	-= formulas			✓		
	Event-handler formulas	✓			✓	
Visualization Characteristics	**1. Relational**	✓		✓	✓	✓
	2. Hierarchal				✓	
	3. Time-Oriented	✓		✓		✓
	4. Interactivity Details on demand	✓				✓
	More or less details	✓			✓	✓
	Filter					✓
	Dynamic Queries					✓

TABLE 2. Examples of Uvis formulas.

	Formula	Description
1.	`Me!Height`	The height of the current visual object
2.	`Height`	Same as formula 1
3.	`Me[index+1]!Height`	The height of a sibling visual object, the next object in the bundle
4.	`Me[5]!Height`	The height of the sixth object in the bundle.
5.	`Parent!Height`	The height of my parent
6.	`ChildLabel!Height`	The height of the first child in the bundle
7.	`TaskBox!Height`	The height of the first TaskBox object
8.	`TaskBox[2]!Height`	The height of the third TaskBox object
9.	`Me.Employee.ID`	The ID field (of Employee table) of the current visual object
10.	`Employee.ID`	same as formula 9
11.	`timeScale!HPos(..)`	The `HPos` function provided by the first instance of a time scale
12.	`index=5 ? Color.Red : Color.Black`	A conditional formula that will evaluate to "red" for the sixth object in the bundle, and "black" to other objects.

The `Rows` *formula* is as expressive as SQL statements, but it is much more compact. For instance, it does not include the `SELECT` clause and the specifications of `JOIN` statements. For instance, consider the formulas for the `Rows` and `Text` properties:

```
Rows: Patient -< ptNote
Text: Title
```

This formula is translated into the following SQL statement:

```
SELECT ptNote.Title FROM Patient
LEFT JOIN ptNote
ON Patient.ptID=ptNote.ptID
```

Uvis compiler only selects the fields that are used by the formulas. In this case, only `ptNote.Title` was selected because it was used in the `Text` property formula. Further, the `ON` specifications were extracted from the data map (a file that describes the primary and foreign keys of tables).

Some advanced visualizations need sorting and grouping data transformations. Such transformations would require designers to use SQL-like operators such as `Order By` and `Group By`. Designers would need to learn such skills prior to attempting such advanced visualizations.

The visual property formulas are expressions similar to spreadsheet formulas, but they can refer to utility functions, visual properties, visual object functions, and fields of any visual object connected to data with a different `Rows` formula. For a complete reference of Uvis formulas, see the Uvis reference card [36].

Utility functions resemble Visual Basic and spreadsheet functions. For instance, the regular math and aggregation functions are available.

Visual objects provide functions the formulas can call. For instance, formulas can call the `HPos` function of `HTimeScale` objects.

Despite the expressive power of Uvis formulas, they have the inherent limitations of declarative formulas. For example, Uvis formulas alone do not support visualizations that require recursive algorithms. Such algorithms contain loops and/or functions that call themselves recursively until a condition is met. Uvis formulas support recursion if it is within the context of existing visual objects. Consider the following formula for a designer Property `TotalHeight`:

```
index=0? Height : Me[index-1]
!TotalHeight + Height
```

This is an example of a conditional formula using the ternary operator known from JavaScript and other languages. The formula means if index is equal to zero (first object in the bundle), the value will be the `Height` value. Otherwise, it will be the previous object's `TotalHeight` added to the `Height` values. This results in the sum of `Height` values in a bundle of visual objects. This is an example of recursion that Uvis formulas allow. However, Uvis formulas and primitive visual objects alone cannot support a visualization such as Tree maps which requires a recursive algorithm. A possible solution is to provide a visual object that performs these complex layout algorithms. Pantazos developed a TreeMap visual object that supports a tree map visualization with Uvis formulas [28].

Another limitation of Uvis formulas is the inability to create visual objects recursively: For instance, Uvis formulas fall short if we want to show a recursive tree-like structure, for instance a folder tree. Since Uvis uses SQL-like formulas, it inherits SQL limitations. For instance, it is not possible to

Data Table

Id	Temperature	Date
1	30	2-6-2011
4	27	15-6-2011
3	33	28-6-2011
4	37	8-7-2011
...	...	...

Visualization

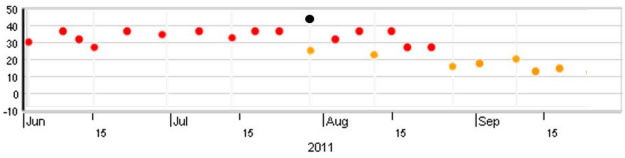


FIGURE 7. (Top) Data behind a custom scatterplot. (Bottom) The custom scatterplot showing temperature reading based on the table at top.

send a query that retrieves the nesting levels of the folders. As a result, it is not possible to create a recursive tree with Uvis formulas because new visual objects must be defined for each level in the tree. A possible solution is to create a specialized visual object that implements the recursive layout.

B. COMPARATIVE ANALYSIS

We selected two popular visualization tools to compare with Uvis: D3 [3] and Vega [12]. The tools were selected because they support custom visualizations, have high citations, and a different approach to visualization construction. For a comparative analysis with other tools, consult [32].

We excluded Protovis [11] as it is the predecessor of D3 [3], and is no longer active. We excluded Vega-lite [13] as it is based on Vega, and is intended for high-level visualization grammar. Moreover, the selected tools were ranked based on the total number of citations on ACM Portal and IEEE website. In a previous work [32], we compared Uvis with other tools such as Prefuse [15] and Improvise [16].

We implemented a custom scatterplot (Figure 7) with the selected tools as well as with Uvis. The scatterplot shows the daily maximum of temperature in a given city in a certain period of time. The circles represent the readings. If the circle is showing the highest temperature in the given period, it is black. Otherwise, if the circle is showing a temperature greater than 25, it is red. The rest of the circles are orange. Although the example is relatively simple, it is a custom scatterplot, and not a standard scatterplot that can be easily made with charting tools. Further, all the tools support it with their language design and visual objects. We were careful not to select a more advanced example as it may be too lengthy and tedious to follow, and it may favor one tool over the others.

1) D3

Figure 8 shows the specifications of this custom scatterplot with D3. The specifications are written as statements that are executed one by one. The maximum temperature variable is defined as it will be used later in the visual encoding (line 1). The visualization basic settings are defined in lines 2-9. D3 uses non-visual scale classes for creating time and numeric axes (lines 10-15). The designer uses the scales to generate x and y axes (lines 16-25). `Circle` objects are defined (lines 26-29). The circles are bound to temperature and date data (lines 30-35). Finally, a conditional expression for the `Fill` property (background color property) sets the color of circles with a conditional expression (line 38).

To sum up, D3 visualization specifications are program-like. Variables and anonymous functions are defined. Further, the instructions are written and executed in sequence. D3 provides non-visual scale classes that facilitate the construction of axes. The axes are not defined directly. Instead, primitive `axis` objects are used for drawing the axes. This separation increases flexibility (e.g designers might define a custom axis in this way), but increases the steps of such a common task. D3 uses declarative expressions to specify the `Circle` visual objects.

2) VEGA

Figure 9 shows the specifications of the same custom scatterplot built with Vega. First, the visualization basic settings are defined in lines 1-7. The data source is defined in lines 10-12. Since we need specific appearance for circles showing the maximum temperature, we need to perform data transformations to calculate the maximum (lines 14-19). Like D3, Vega uses non-visual scale classes for creating time and numeric axes (lines 22-34). The designer uses the scales to generate x and y axes (lines 35-37). The designer specifies circles and bind them to temperature and date data (lines 38-46). Finally, a conditional expression for the `Fill` property (background color property) sets the color of circles based on the logic we discussed earlier (line 47).

Vega uses a declarative approach to visualization specifications. Unlike D3, Vega does not require the specifications to be written in a certain sequence. Vega requires data transformations to be explicitly specified for extracting aggregate functions such as max, min, sum, and average. Like D3, Vega uses non-visual scale classes that facilitate the construction of axes. The visual axes are defined separately. To define a formula-like expression, designers need to use *a signal*, a dynamic variable that parameterizes a visualization.

3) UVIS

Figure 10 shows the specifications of the custom scatterplot with Uvis. To create the time and numeric axes, the designer dragged `HTimeScale` and `VNumericScale` visual objects from the toolbox and dropped them on a form. The designer moved and resized them until they looked satisfactory. Uvis sets position properties (i.e. `Top`, `Height`, etc.) accordingly. To define the range of time and numbers the scales show, the designer typed the value of the `BorderValues` property in the property grid (lines 5 and 11). To create circles representing the

```
a
1. var max_temperature = d3.max(data, d => d.temperature);
2. var margin = {top: 20, right: 20, bottom: 30, left: 40};
3. var width = 500 - margin.left - margin.right;
4. var height = 250 - margin.top - margin.bottom;
5. var svg = d3.select("#chart").append("svg")
6.          .attr("width", width + margin.left + margin.right)
7.          .attr("height", height + margin.top + margin.bottom)
8.          .append("g")
9.          .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
b
10.var x = d3.time.scale()
11.         .domain([new Date(2020, 2, 1), new Date(2020, 2, 26)])
12.         .range([0, width]);
13.var y = d3.scale.linear()
14.         .domain([30, -10])
15.         .range([0, height]);

16.var xAxis = d3.svg.axis()
17.         .scale(x);

18.var yAxis = d3.svg.axis()
19.         .scale(y)
20.         .orient('left');
21.svg.append("g")
22.         .attr('transform', 'translate(0,' + height + ')')
23.         .call(xAxis);
24.svg.append("g")
25.         .call(yAxis);
c
26.var circles = svg.selectAll("circle")
27.         .data(data)
28.         .enter()
29.         .append("circle");
d
30.circles.attr('cx', function (d) {
31.             return x(d.date);
32.         })
33.         .attr('cy', function (d) {
34.             return y(d.temperature);
35.         })
36.         .attr('r', 6)
37.         .style('fill', function (d) {
38.             return d.temperature == max_temperature ? "black" : d.temperature >
    25 ? "red" : "orange";
39.         });
```

FIGURE 8. Building a custom scatterplot with D3. (a) defining the visualization. (b) defining the numeric (temperature) and time scales (axes). (c) defining circles. (d) Visually mapping the circles to temperature and date fields according to the scales.

temperature reading, the designer drags and drops an `Ellipse`. The designer bound the `Ellipse` objects to data using the `Rows` formula (line 16). The designer typed formulas for the position properties (`Top` and `Left`) (lines 17 and 18). The formulas call position functions (`HPos`, `VPos`) provided by the scales to calculate the positions.

Unlike D3 and Vega, Uvis uses only visible visual objects. Further, Uvis does not require data transformations to extract aggregate functions (such as max, sum). Designers specify the aggregate functions in the property where they will be used. Like Vega, Uvis uses declarative expressions that directly define the visual properties. Further, there is no need to define variables, and the sequence of specifying the expressions is unimportant. The environment shows the available visual objects, and allows the designers to drag, drop, and resize them (as long as the position and size properties do not have dynamic expressions) rather than textually setting them. Uvis expressions are more expressive than D3 and Vega as they can refer to visual properties of the same and other visual objects. For instance, the `Left` formula of the `TemperatureEllipse` refers to a visual property `Width` as well as a function of a visual object `HPos` (Figure 10).

a
b
c
d

```
1. {
2.   "$schema": "https://vega.github.io/schema/vega/v5.json",
3.   "description": "A custom x-y graph ",
4.   "width": 650,
5.   "height": 300,
6.   "padding": 5,
7.   "autosize": {"type": "fit", "contains": "padding"},
8.   "data": [
9.     {
10.      "name": "temperature",
11.      "url": "data.csv",
12.      "format": {"type": "csv", "parse": {"temperature":"number", "date":"date"}}
13.            ,
14.      "transform": [
15.         { "type": "joinaggregate",
16.           "fields": ["temperature"],
17.           "ops": [ "max"],
18.           "as": [ "max_temp"]
19.           }]
20.    }
21.    ],
22.    "scales": [
23.    {
24.      "name": "x",
25.      "type": "time",
26.      "domain": {"data": "temperature", "field": "date"},
27.      "range": "width"
28.    },
29.    {
30.      "name": "y",
31.      "type": "linear",
32.      "domain": {"data": "temperature", "field": "temperature"},
33.      "range": "height"
34.    }
35.  ],"axes": [
36.    {"orient": "left", "scale": "y"},
37.    {"orient": "bottom", "scale": "x"}
38.  ],"marks": [
39.    { "name": "marks",
40.      "type": "symbol",
41.      "from": {"data": "temperature"},
42.      "encode": {
43.        "enter": {
44.          "x": {"scale": "x", "field": "date"},
45.          "y": {"scale": "y", "field": "temperature"},
46.          "shape": {"value": "circle"},
47.          "fill": {"signal": "datum['temperature'] == datum['max_temp']? 'black':
datum['temperature']   > 25 ? 'red' : 'orange'"}
48.      }
49.        }
50.      }
51.
52.      ]}
```

FIGURE 9. Building a custom scatterplot with Vega. (a) defining the visualization. (b) defining the data transformations. (c) defining the numeric (temperature) and time scales (axes). (d) defining circles and visually mapping them to temperature and date fields according to the scales.

C. EVALUATION STUDIES

We conducted several evaluation studies. Our objectives were to evaluate the learnability of Uvis and identify the concepts that are not easy to learn. Here we show the details of one evaluation study with seven participants. All the participants were non-programmers. They had no prior knowledge of the Uvis formulas, and had never used the Uvis environment. They had basic knowledge of Excel

```
a
1.  Form: form
2.  Width:600
3.  Height:250

b
4.  HTimeScale: tScale
5.  BorderValues: 2011-06-01, 2011-10-01
6.  Canvas: canvas
7.  Width: 530
8.  Top: 150
9.  Left: 10

10. VNumericScale: nScale
11. BorderValues: -10, 50
12. Orientation: Orientation.Right
13. Height: 90
14. Top: 40
15. Left: 10

c
16. Ellipse: TemperatureEllipse
17. Rows: HighTemperature
18. Top: nScale!VPos(Temperature) - Height/2
19. Left: tScale!HPos(Date) - Width/2
20. BackColor: MAX(Temperature)=Temperature ? Color.Black :
     Temperature >25? Color.Red : Color.Orange
```

FIGURE 10. Building a custom scatterplot with Uvis. (a) The visualization form. (b) The horizontal and vertical scales (axes). (c) The ellipses (circles) showing the data.

TABLE 3. Profiles of participants of evaluation studies.

Profile	Participants						
	1	2	3	4	5	6	7
Position	volunteer	student	student	businessman	student	student	Loan manager
Know what a database table is	✓	✓	✓	✓	✓	✓	✓
Familiar with ER models	✓					✓	✓
Familiar with basic algebraic equations	✓	✓	✓	✓	✓	✓	✓
Familiar with basic math sequences	✓	✓	✓	✓	✓	✓	✓
Familiar with arithmetic spreadsheet formulas (e.g. +, *)	✓	✓	✓	✓	✓	✓	✓
Familiar with spreadsheet aggregate functions (e.g. SUM, MAX, AVG)			✓				
Familiar with logical expressions	✓		✓				
Knows how to interpret simple visualizations (e.g. line chart, pie chart)	✓	✓	✓		✓	✓	✓

formulas, algebra, trigonometry, and sequences, and knew what a database table is. Further, they knew how to read simple visualizations such as bar charts. Table 3 shows the participant profiles in more detail.

We judged that the number of participants we tested with was sufficient for the purpose of identifying whether the Uvis concepts were understandable. Nielsen *et al.* empirically found out that most of the usability problems are found by testing with five users [30].

1) METHODOLOGY

Each evaluation study lasted 2 hours on average. The studies were carried out in a lab. Each participant viewed two screens. One screen showed a Microsoft PowerPoint-based step-by-step tutorial available, and the other showed the Uvis environment. Each participant was asked to follow the instructions in the tutorial. The tutorial is divided into sections, at the end of which, designers were given a task to work on their own, but they could go back to the tutorial and/or example solutions. The tutorial can be found in [48].

The participants were asked to think aloud while they were carrying out the tasks. They received no help from us during the study.

To evaluate ease of learning, we measured task completion time (T) and the quality of the solution (Q). The quality of the solution was measured by comparing the participant's solution against the optimal solution and then rating it on a scale 0-10.

To find out which concepts are easy or hard to understand, and collect other information related to Uvis, we observed the participants while they used the tool, and asked them to provide feedback at the end of the evaluation study. The detailed documentation can be found at [27].

2) TEST TASKS

Figure 11 shows the test tasks the participants carried out in the evaluation. The tasks are modification tasks. Customizing a visualization as opposed to building one from the ground up is less time consuming and challenging to designers because they have something to build on. This is adequate for our purpose of identifying Uvis concepts that are hard to understand. The participants were given the desired visual output, a given visualization, as well as a written description of the requirements. The participants could ask for clarifications.

- **Task 1:** The bars show a company's monthly sales. Position the bars representing monthly sales like a horizontal list, make the bar heights represent the monthly sales, and order them based on the sales.
- **Task 2:** The ellipses on top show all runners in a marathon. The ones on the bottom show runners that are citizens. For the ellipses on the top, make the male runners blue, and the female runners pink. For the ellipses at the bottom, show only runners older than 30.
- **Task 3:** A pie chart shows several classes of passengers (e.g. crew, emperor, etc.). The male passengers are shown on the top as light blue pie slices. Show female passengers on the top as pink pie slices.
- **Task 4:** The red curves represent the high readings of the weather in three cities in a period of time. Show the low readings as blue lines.

3) SURVEY QUESTIONS

At the end of the study, the participants were asked to fill out a survey. The purpose of the survey was to identify weaknesses in Uvis concepts that would be basis of improvement in future releases.

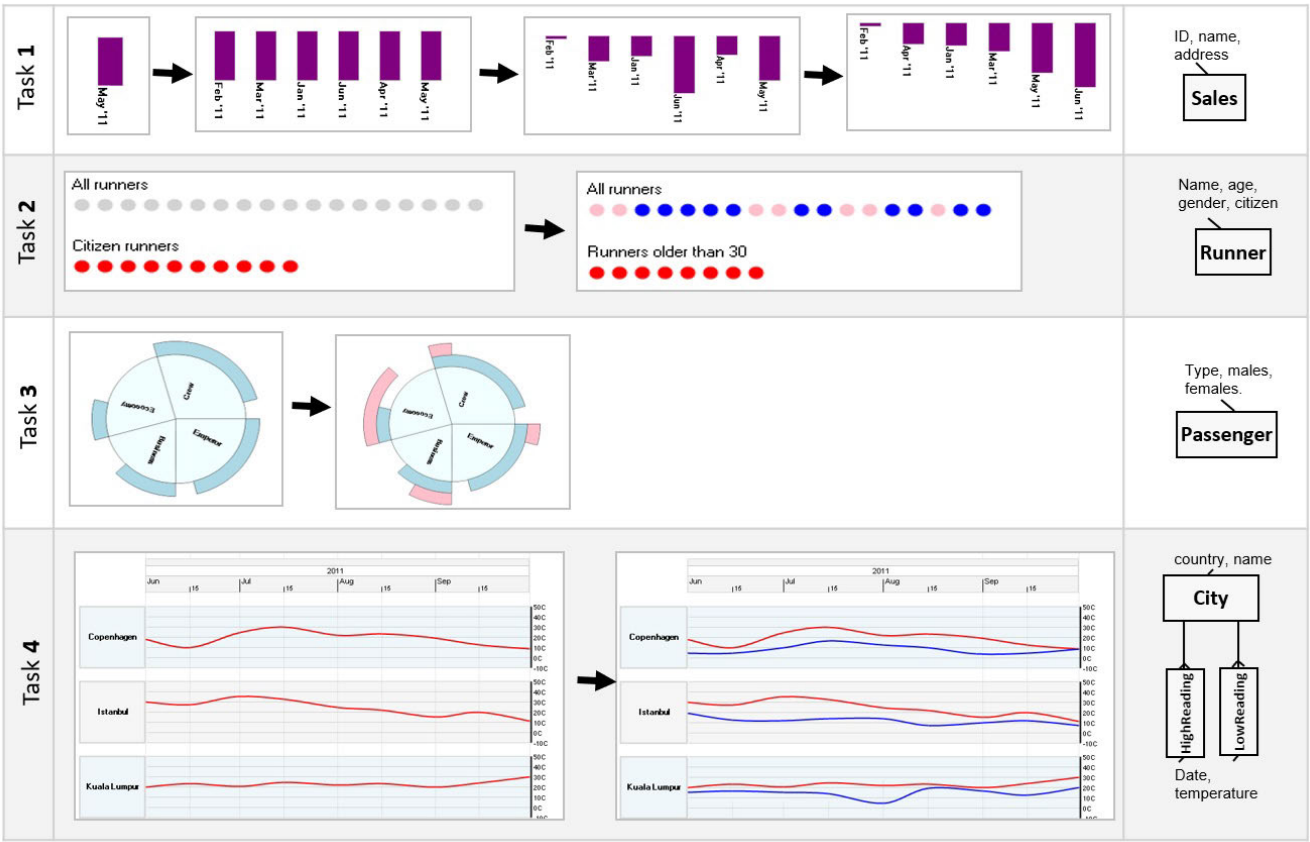


FIGURE 11. (Left) Test tasks of usability studies: The presented visualizations and the required modifications. (Right) Data tables behind the visualizations.

TABLE 4. Quantitative results of the evaluation studies.

	Participants							Summary	
	1	2	3	4	5	6	7	Time (m)	Quality (0-10)
Task 1	T: 12.4 Q: 10	T: 5.5 Q: 10	T: 7.6 Q: 10	T: 14 Q: 7	T: 25 Q: 7	T: 23 Q: 4	T: 25 Q: 7	μ : 16 σ : 7.6	μ : 7.8 σ : 1.7
Task 2	T: 8.3 Q: 8	T: 4 Q: 10	T: 4.7 Q: 9	T: 8.1 Q: 5	T: 14 Q: 5	T: 12 Q: 8	T: 14 Q: 5	μ : 9.3 σ : 3.8	μ : 7.1 σ : 1.9
Task 3	T: 6.6 Q: 10	T: 15.3 Q: 9	T: 9.3 Q: 10	T: 23 Q: 4	T: 14 Q: 5	T: 12 Q: 2	T: 14 Q: 5	μ : 13.4 σ : 4.8	μ : 6.4 σ : 2.9
Task 4	T: 4.6 Q: 9	T: 7.6 Q: 9	T: 6.4 Q: 10	T: 11 Q: 5	T: 13 Q: 9	T: 12 Q: 2	T: 13 Q: 9	μ : 6.6 σ : 3.1	μ : 7.5 σ : 2.7

We asked the participants about their experience in general, and any difficulties they encountered during the study. Furthermore, we asked about formulas that they thought were hard to understand.

To evaluate understandability of formulas, we asked participants about the functionality of some operators such as "!", ".", "-<", and "index".

4) RESULTS

Table 4 shows the quantitative results. The participants were able to complete the tasks at different times. They managed to complete most of the requirements. Despite the variability in quality score and time, the results are encouraging. Task 3 had the lowest average quality score (6.4). This might be described by designers needing to understand unfamiliar formulas such as the formula of `SweepAngle` in the `PieSlice`. Nevertheless, half of the participants successfully implemented task 3 as they saw the similarity between the blue pie slice and the pink pie slice. The longest time a participant spent on a task was the time participant 7 spent on the first task (25 minutes). This might be explained by the number of modifications needed for the task (three different modifications). Nevertheless, the time the participant needed was still within the total time we had planned for.

The qualitative results can be summarized as follows: The participants were able to learn the `Rows` formulas that are used to connect the visual objects to data. All participants were able to explain the "-<" operator. Additionally, the participants found basic visual property formulas easy to understand. However, half of the participants were confused about the difference between the dot operator (.) and the bang operator (!).

We observed that all participants used most of the Uvis environment components to work on tasks. In particular, participants used the data model, data view, and property grid. Participant 2 appreciated that she viewed everything needed

to complete the task at hand. When asked after the end of each task about how confident they are about their solution, two participants inspected their solution to check the visual mappings and answered ''yes''.

V. CONCLUSION AND FUTURE WORK

This paper presented Uvis, a visualization tool that targets end-user developers without programming skills. With Uvis, designers drag and drop visual objects, set visual properties with formulas, and see the result immediately. The formulas are declarative and similar to spreadsheet formulas. The formulas compute the property values and can refer to fields, visual properties, functions, etc. Cognitive aids assist designers while implementing a visualization. Uvis formulas can express custom visualizations that are made of primitive and specialized visual objects. Since they are declarative, Uvis formulas do not support visualizations with recursive layout. As a remedy, a specialized object will have to be made for that purpose. Our evaluation shows that designers can learn the basics of Uvis relatively fast, and can customize visualizations. Based on the evaluation studies we conducted, we have improved autocompletion of formulas as well as error messages to help designers understand Uvis better.

Currently Uvis is desktop based. We are working on making Uvis available on several platforms including web and mobile. Uvis currently supports raw relational data. Many commercial systems hide their data behind web-services and multi-layer architectures, and are unable to give access to data in such a way that end-user developers can join tables and filter them according to end-user needs. We are currently exploring accessing the data with OData [49], which can take an SQL statement as a parameter.

REFERENCES

[1] J. Bertin, *Semiology of Graphics: Diagrams Networks Maps*. Madison, WI, USA: Univ. Wisconsin Press, 1983.

[2] C. Stolte, D. Tang, and P. Hanrahan, ''Polaris: A system for query, analysis, and visualization of multidimensional relational databases,'' *IEEE Trans. Vis. Comput. Graphics*, vol. 8, no. 1, pp. 52–65, Jan./Mar. 2002.

[3] M. Bostock, V. Ogievetsky, and J. Heer, ''D^3 data-driven documents,'' *IEEE Trans. Vis. Comput. Graphics*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011, doi: 10.1109/TVCG.2011.185.

[4] C. Williamson and B. Shneiderman, ''The dynamic HomeFinder: Evaluating dynamic queries in a real-estate information exploration system,'' in *Proc. 15th Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retr.*, 1992, pp. 338–346.

[5] A. Satyanarayan and J. Heer, ''Lyra: An interactive visualization design environment,'' in *Proc. Eurograph. Conf. Vis. (EuroVis)*, 2014, vol. 33, no. 3, p. 10.

[6] D. Ren, T. Hollerer, and X. Yuan, ''iVisDesigner: Expressive interactive design of information visualizations,'' *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 12, pp. 2092–2101, Dec. 2014, doi: 10.1109/TVCG.2014.2346291.

[7] H. Mei, W. Chen, Y. Ma, H. Guan, and W. Hu, ''VisComposer: A visual programmable composition environment for information visualization,'' *Vis. Inform.*, vol. 2, no. 1, pp. 71–81, Mar. 2018.

[8] H. Xia, N. H. Riche, F. Chevalier, B. De Araujo, and D. Wigdor, ''DataInk: Direct and creative data-oriented drawing,'' in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2018, pp. 1–13, Paper 223, doi: 10.1145/3173574.3173797.

[9] D. Ren, B. Lee, and M. Brehmer, ''Charticulator: Interactive construction of bespoke chart layouts,'' *IEEE Trans. Vis. Comput. Graphics*, vol. 25, no. 1, pp. 789–799, Jan. 2019.

[10] A. Inselberg, ''The plane with parallel coordinates,'' *Vis. Comput.*, vol. 1, no. 2, pp. 69–91, Aug. 1985.

[11] M. Bostock and J. Heer, ''Protovis: A graphical toolkit for visualization,'' *IEEE Trans. Vis. Comput. Graphics*, vol. 15, no. 6, pp. 1121–1128, Nov. 2009.

[12] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer, ''Reactive vega: A streaming dataflow architecture for declarative interactive visualization,'' *IEEE Trans. Vis. Comput. Graphics*, vol. 22, no. 1, pp. 659–668, Jan. 2016.

[13] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, ''Vega-lite: A grammar of interactive graphics,'' *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 1, pp. 341–350, Jan. 2017.

[14] D. Park, S. M. Drucker, R. Fernandez, and N. Elmqvist, ''Atom: A grammar for unit visualizations,'' *IEEE Trans. Vis. Comput. Graphics*, vol. 24, no. 12, pp. 3032–3043, Dec. 2018.

[15] J. Heer, S. K. Card, and J. A. Landay, ''Prefuse: A toolkit for interactive information visualization,'' in *Proc. CHI*, 2005, pp. 421–430.

[16] C. Weaver, ''Building highly-coordinated visualizations in improvise,'' in *Proc. INFOVIS*, 2004, pp. 159–166.

[17] D. A. Norman, *User Centered System Design: New Perspectives on Humancomputer Interaction*. Boca Raton, FL, USA: CRC Press, 1986.

[18] B. Myers, S. E. Hudson, and R. Pausch, ''Past, present, and future of user interface software tools,'' *ACM Trans. Comput.-Hum. Interact.*, vol. 7, no. 1, pp. 3–28, Mar. 2000.

[19] L. Grammel, M. Tory, and M. D. Storey, ''Erratum to 'how information visualization novices construct visualizations,''' *IEEE Trans. Vis. Comput. Graphics*, vol. 17, no. 2, p. 260, Feb. 2011.

[20] J. Heer, F. Ham, S. Carpendale, C. Weaver, and P. Isenberg, ''Creation and collaboration: Engaging new audiences for information visualization,'' in *Information Visualization* (Lecture Notes in Computer Science), vol. 4950, A. Kerren, J. Stasko, J.-D. Fekete, and C. North, Eds. Berlin, Germany: Springer, 2008, pp. 92–133.

[21] W. S. Cleveland, *The Elements of Graphing Data*. Hobart, TAS, Australia: Hobart Press, 1994.

[22] L. Grammel, C. Bennett, M. Tory, and M. Storey, ''Survey of visualization construction user interfaces,'' in *Proc. Eurograph. Conf. Vis. (EuroVis)*, 2013, pp. 1–5.

[23] P. P.-S. Chen, ''The entity-relationship model—Toward a unified view of data,'' *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, Mar. 1976, doi: 10.1145/320434.320440.

[24] Z. Liu, J. Thompson, A. Wilson, M. Dontcheva, J. Delorey, S. Grigg, B. Kerr, and J. Stasko, ''Data illustrator: Augmenting vector design tools with lazy data binding for expressive visualization authoring,'' in *Proc. CHI*, Montreal, QC, Canada, Apr. 2018, pp. 1–13.

[25] C. Plaisant, B. Milash, A. Rose, S. Widoff, and B. Shneiderman, ''LifeLines: Visualizing personal histories,'' in *Proc. CHI*, 1996, pp. 221–227.

[26] B. Shneiderman, ''The eyes have it: A task by data type taxonomy for information visualizations,'' in *Proc. IEEE Symp. Vis. Lang.* Washington, DC, USA: IEEE Computer Society Press, 1996, pp. 336–343.

[27] M. A. Kuhail, ''Custom formula-based visualizations for Savvy designers,'' Ph.D. dissertation, Dept. Softw. Syst. Sect., IT Univ. Copenhagen, Copenhagen, Denmark, 2013.

[28] K. Pantazos, ''Custom data visualization without real programming,'' Ph.D. dissertation, Dept. Softw. Syst., IT-Univ. København, København, Denmark, 2013.

[29] D. A. Keim, J. Schneidewind, and M. Sips, ''CircleView: A new approach for visualizing time-related multidimensional data sets,'' in *Proc. Work. Conf. Adv. Vis. Interfaces (AVI)*, New York, NY, USA, 2004, pp. 179–182.

[30] J. Nielsen and T. K. Landauer, ''A mathematical model of the finding of usability problems,'' in *Proc. ACM INTERCHI Conf.*, Amsterdam, The Netherlands, 1993, pp. 206–213.

[31] H. Mei, Y. Ma, Y. Wei, and W. Chen, ''The design space of construction tools for information visualization: A survey,'' *J. Vis. Lang. Comput.*, vol. 44, pp. 120–132, Feb. 2018.

[32] M. A. Kuhail, S. Lauesen, K. Pantazos, and X. Shangjin, ''Usability analysis of custom visualization tools,'' in *Proc. SIGRAD Interact. Vis. Anal. Data*, Växjö, Sweden, Nov. 2012, pp. 19–28.

[33] M. A. Kuhail, K. Pandazo, and S. Lauesen, ''Customizable time-oriented visualizations,'' in *Proc. Int. Symp. Vis. Comput.* Berlin, Germany: Springer, 2012, pp. 668–677.

[34] M. A. Kuhail, S. Lauesen, and K. Pantazos, ''The inspector: A cognitive artefact for visual mapping,'' in *Proc. IVAPP*, Feb. 2013, pp. 1–10.

[35] K. Pantazos, M. A. Kuhail, S. Lauesen, and S. Xu, "uVis studio: An integrated development environment for visualization," in *Proc. Vis. Data Anal.*, Feb. 2013, pp. 15–30.
[36] S. Lauesen. (Apr. 2020). *Uvis Reference Card V2.3*. [Online]. Available: http://www.itu.dk/people/slauesen/S-EHR/UvisCard.pdf
[37] *Uvis Trial Version*. Accessed: Feb. 2020. [Online]. Available: https://www.itu.dk/~slauesen/UvisTrial_Latest.zip2009-2020
[38] *GDI+*. Accessed: Feb. 2020. [Online]. Available: https://rb.gy/bdwj2q
[39] (2012). *Java2D*. Accessed: Sep. 2019. [Online]. Available: https://docs.oracle.com/javase/tutorial/2d/index.html
[40] M. Mauri, T. Elli, G. Caviglia, G. Uboldi, and M. Azzi, "RAWGraphs: A visualisation platform to create open outputs," in *Proc. ACM Italian CHI Conf.*, 2017, pp. 28:1–28:5, doi: 10.1145/3125571.3125585.
[41] J. Heer, N. Kong, and M. Agrawala, "Sizing the horizon: The effects of chart size and layering on the graphical perception of time series visualizations," *ACM Hum. Factors Comput. Syst.*, 2009, pp. 1303–1312.
[42] *Flourish*. Accessed: Apr. 2020. [Online]. Available: https://flourish.studio
[43] *Infogram*. Accessed: Apr. 2020. [Online]. Available: https://infogram.com
[44] *Tableau*. Accessed: Feb. 2020. [Online]. Available: https://www.tableau.com/
[45] *Spotfire*. Accessed: Feb. 2020. [Online]. Available: https://www.tibco.com/products/tibco-spotfire
[46] *Omniscope*. Accessed: Feb. 2020. [Online]. Available: http://www.visokio.com/omniscope
[47] *Florence Nightingale's Rose Diagram*. Accessed: Feb. 2020. [Online]. Available: http://www.historyofinformation.com/detail.php?entryid=3815
[48] M. A. Kuhail, "Uvis documentation (version V3)," Zenodo, Tech. Rep., Apr. 2020, doi: 10.5281/zenodo.3865081.
[49] *OData Documentation*. Accessed: Feb. 2020. [Online]. Available: https://docs.microsoft.com/en-us/odata/

MOHAMMAD AMIN KUHAIL received the M.Sc. degree in software engineering from the University of York, in 2006, and the Ph.D. degree in computer science from the IT University of Copenhagen, Denmark, in 2013. He has served as an Assistant Teaching Professor with the University of Missouri–Kansas City, USA, for six years. In 2019, he joined Zayed University, United Arab Emirates , where he is currently serves as an Assistant Professor. He is also a Computer Scientist and a Software Engineer with a diverse skill set that spans web development, object-oriented programming, algorithms, usability, and data science. His research interests include end-user development, usability analysis, and computer science education.

SOREN LAUESEN received the M.Sc. degree in mathematics and physics from the University of Copenhagen, Denmark, in 1965, and the B.Com. degree from the Copenhagen Business School, Denmark, in 1979. From 1962 to 1973, he worked as a Developer/Department Manager with Regnecentralen, Denmark (Danish computer manufacturer). From 1969 to 1972, he was a part-time Associate Professor with the University of Copenhagen, and a Co-Founder of the first computer science education in Denmark. From 1973 to 1976, he was also a Co-Founder of the Software Development Department, Brown Boveri, Copenhagen (now ABB). From 1976 to 1979, he was a Visiting Professor with the University of Copenhagen, and the Department Manager for the last two years. From 1979 to 1985, he was also a Co-Founder of the Software Development Center, NCR, Copenhagen. From 1985 to 1999, he was a Professor with the Copenhagen Business School, and a Co-Founder of the combination education in business and computer science. He has served as the Head of the Department, from 1992 to 1996. In 1999, he became a Professor with the IT University of Copenhagen, where he has served for 20 years. Since September 2019, he has been a Professor Emeritus with the IT University of Copenhagen.